Assertion Support for Manipulating Constrained Data-Centric XML

Patrick Michel
University of Kaiserslautern, Germany
p_michel@cs.uni-kl.de

Arnd Poetzsch-Heffter
University of Kaiserslautern, Germany
poetzsch@cs.uni-kl.de

ABSTRACT

XML is used for different purposes. We are interested in data-centric applications of XML where it is used to handle structured data in loosely coupled, distributed systems. In many such scenarios, it is important that the XML data complies to structural and integrity constraints, in particular to value-based constraints. The constraints should remain invariant under operations manipulating the data.

In this paper, we present a technique to maintain the invariants. For a core procedural manipulation language, we show how to automatically derive the weakest precondition of procedures for the constraints. We present a formalization of a data-centric XML abstraction and an assertion language that enables weakest precondition generation. Our framework allows to automatically simplify and reduce the generated preconditions such that checking becomes independent of the size of the constraint definitions. This is achieved by isolating the aspects that are manipulated by the considered procedure from the aspects that remain unchanged. As the overall invariant can be assumed for the input data, only the isolated aspects have to be checked in prestates of procedures.

1. INTRODUCTION

XML is used for different purposes. A well-known distinction is between document-centric and data-centric use. In *document-centric* XML, a text is annotated with markup, representing additional information, which is not essential to its original meaning.

Our research targets *data-centric XML* used to represent tree-shaped data. While the structure of this tree is essential to its meaning, the ordering of siblings is often insignificant. The use of text is reduced to simple string values and the content of data-centric XML documents mainly consists of arithmetic and enumeration values.

XML can be used as a dynamic language, where both form and content aren't fixed and can change often. Unlike such *schema-less* uses of XML, we are interested in *schema-aware* applications. In many service-oriented architectures, it is crucial to specify data constraints concerning form and content and maintain these contraints as invariants of the operations processing the XML data.

Approaches like regular expression types [18] are powerful enough to allow the derivation of static guarantees, yet such technologies especially support the correct transformation of XML. In contrast, we are interested in maintaining invariants in form of *fixed* schemata and assume only sequential *local* manipulations. We target systems whose data model is *designed* rather than evolved by *prototyping* and which have a domain-specific set of atomic local update procedures.

Structural invariants, and also integrity constraints, are commonly used in the field of XML. Languages like XML Schema [23, 24] or Relax NG [12] allow their specification in a grammar-based approach. We are also interested in *value-based* invariants, which constrain the actual values of documents.

Languages like Schematron allow their specification using XPath expressions in a rule-based approach. In contrast to structural constraints, however, it is in general far more difficult to guarantee that a procedure manipulating an XML document will maintain such constraints.

The main contribution of this paper is a formalization of a data-centric XML abstraction, which allows to

- specify structural, integrity and value-based constraints
- capture the semantics of local manipulation procedures
- express and automatically derive weakest preconditions for procedures
- drastically simplify preconditions and enables to maintain invariants at low costs

All this is achieved by carefully designing the formalization and restricting the amount of generic programming. The formalization focuses on paths, which are both used to describe documents as sets of paths and to reason about local tree updates. Invariants for XML can become very large, while local manipulations often affect only a small portion of it. By analyzing and thereby isolating affected and unaffected parts of the invariant using structural arguments, it is possible to considerably reduce runtime efforts to maintain invariants.

Section 2 will give an example of how this approach works. The formalization of the used XML abstraction is then described in Section 3. The section also presents our main theorem about assertion simplification. Section 4 contains

Copyright is held by the author/owner.
International Workshop on Programming Language Techniques for XML (PLAN-X 2009), January 24, Savannah, Georgia

the definition of a core procedural language based on the formalization. In addition, the section introduces the concept of invariants and shows how they are maintained. The necessary lemmata to prove the theorems of Section 3 are presented in Section 5, which also provides more details of the formalization. We implemented the approach in a prototype system and discuss the practical concerns in Section 6, where we also discuss the size of resulting preconditions. Related work is discussed in Section 7.

2. MOTIVATION

In this section, we will give an example to illustrate the application background of our approach. As explained in the introduction, XML documents are considered to be representations of tree-structured data. We will specify an XML data format, as well as a small procedure modifying documents of that format. We will then outline the desired results of static analysis.

2.1 Constraints

In our example specification language, the structure of a document is specified using a grammar-based approach, using a syntax very similar to abbreviated Relax NG [11, 12]. Integrity constraints, and also value-based constraints, are specified using a rule-based approach in an expression language resembling XPath. The rules are embedded in the form of *constraints* in the structural pattern, using brackets. Their location in an element or attribute is significant, as we refer to the current one with the dot.

```
start =
  element inventory {
    attribute time      { integer [. ≥ 0] },
    attribute capacity  { integer [. > 0] },

    [ ./capacity ≥ sum (./type[./item*/typeref]/size) ]

    element type* {
      attribute size    { integer [. > 0] [. ≤ //capacity] }
    },

    element item* {
      attribute since   { integer [. ≥ 0] [. ≤ //time] },
      attribute typeref { ident [ //type[.] ∈ $ ] }
    }
  }
```

The specification demands that valid documents contain an `inventory`, which always has the attributes `time` and `capacity`. The `time` represents a global time stamp, which should always be a natural number, and `capacity` is a positive natural number with the obvious meaning. An inventory can contain an arbitrary number of `items` and item `types`.

To distinguish different `items` or `types` and select single ones, each element in a document has an identifier. We assume it to be stored in a special attribute named `id`, which does not explicitly appear in structural specifications like the above. In contrast to XPath, these identifiers are not globally unique, but only locally, for every kind of element. For example, two different `items` must not have the same identifier, but an `item` and a `type` may. Elements which can appear only once always use the special identifier *null*.

An item `type` just has a `size` attribute, which is a positive natural number not exceeding the inventory capacity. Each `item` has a time stamp called `since`, which represents the time it was inserted into the inventory and should consequently be a natural number not exceeding the current global `time`.

Our approach also supports references of elements within a document. For example, each `item` contains an attribute `typeref`, which references the `type` of the `item`. The above specification expresses that the `types` referenced by `items` must be contained in the document. To refer to the *current document* we use the $ symbol.

As an overall constraint, we want to make sure that the summation of all item `sizes` currently in the inventory does not exceed its capacity. This constraint is specified after the attribute `capacity` and compares its value to the summation of `type sizes`, which are referenced from `items`.

2.2 Updates and Invariants

To add an `item` of a given `type` to the inventory, we use the procedure `addItem`:

```
proc addItem(ident itemId, ident typeId) {
  append /inventory
    <item @id=[itemId] @since=[//time] @typeref=[itemId]/>
}
```

The procedure takes a document for which all constraints hold as an implicit argument. In addition, it takes a unique identifier for the `item` as well as one for its `type`. The procedure then just appends a new `item` given as XML fragment to the inventory. We use the parameter `itemId` as identifier for the `item`, whereas `typeId` is used as identifier reference, so we can refer to the item `type`. The `since` attribute is set to the current `time` of the inventory.

The central question we want to address in this paper is: What are suitable preconditions for such procedures such that specified constraints are maintained. Given any document satisfying the specified constraints and any set of actual parameters for the procedure: Will the resulting document still satisfy the constraints?

In a traditional approach we would need to check the specification against the resulting document and hope for the best. If the check fails we would have to roll back the changes and it is unclear what to do instead.

Ideally we would like to check only the *relevant* parts of the specification, *without* having executed the procedure, i.e. on the initial document. In fact, for the procedure `addItem`, we need to check only three preconditions, which are necessary as they depend on the input document and both parameters:

```
/inventory/item[itemId] ∉ $
/inventory/type[typeId] ∈ $
/inventory/capacity ≥ /inventory/type[typeId]/size +
  sum (/inventory/type[/inventory/item*/typeref]/size)
```

First the `item` we insert must not already exist in the document, as this would violate the uniqueness of `itemId`. The opposite has to be true for `typeId`, as we are referring to a `type` using this identifier, which must exist. But most importantly, inserting the `item` changes the total item size, so we need to check if the `capacity` is exceeded. If we have the sum of the old item sizes cached, we just need to add the new one and compare it to the `capacity`.

These are the only constraints that can be violated by the procedure, nothing additional has to be checked. Note that many of the constraints in the specification have to hold for *every* `item` or `type`, yet the precondition does not mention all these other `items` and `types` at all, as we are not manipulating them. Also note that there is no need to check the constraints applying to the attribute `since`. It is obviously smaller or equal to `time`, as we are setting it to `time`, and it is also always a natural number, as `time` is known to be a natural number.

A computer should be able to automatically derive which parts of a specification are necessary to check, so a developer does not need to worry about these things. It should even be possible to do so automatically before executing the procedure. If a precondition fails, there is a concrete reason for this and the programmer can most often react to the failure, correct it and invoke the procedure with different parameters or even another procedure.

2.3 Maintaining Invariants

In this paper we show how constraints can be be maintained as invariants, without the need to give a specification for procedures. The postcondition of every procedure P is the constraint C given as invariant. We are able to automatically generate weakest preconditions and from there the necessary runtime checks. An invariant can be efficiently maintained by doing only minimal incremental checks.

For the invariant to hold in the poststate of a procedure, it is sufficient that the weakest precondition $wp(P, C)$ is true on the initial document \$. $wp(P, C)$ expresses C and additional properties related to the parameters and document changes. As C is an *invariant*, however, we assume it to hold in the prestate of the procedure and we can significantly weaken the precondition to $C \rightarrow wp(P, C)$. This, by itself, does not change the effort necessary to check, as we know C to hold and we are again left with $wp(P, C)$.

So the goal is to normalize $wp(P, C)$ and split it up into two parts

$$wp(P, C) \Leftrightarrow C_w \wedge C_s$$

where C_w is a weakened form of the original invariant C, i.e. $C \rightarrow C_w$ holds, and C_s is the isolated part of the invariant which was subject to substitution and which is not implied by the invariant C. We can then do the following reduction:

$$\begin{aligned} C \rightarrow wp(P, C) &\Leftrightarrow C \rightarrow (C_w \wedge C_s) \\ &\Leftrightarrow (C \rightarrow C_w) \wedge (C \rightarrow C_s) \\ &\Leftrightarrow \mathit{true} \wedge (C \rightarrow C_s) \\ &\Leftrightarrow C \rightarrow C_s \end{aligned}$$

By assuming C to hold, maintaining an invariant now reduces to checking C_s. Most often, C_s is of much smaller size than C. In fact, adding more constraints to an invariant, which are not affected by a procedure, does not affect the size of C_s at all.

In the following sections, we formalize an abstraction of data-centric XML, which allows automated derivation of C_s. In fact, our implementation described in Section 6 derives exactly the shown precondition, when given the example specification and procedure. The formalization focuses on static analysis and consequently restricts forms of generic programming, which make it impossible to verify properties statically and increase efforts at runtime. We focus on a set of XML core features that can be handled completely automatic, resulting in very concise weakest preconditions.

3. ASSERTION LANGUAGE

In this section, we formalize a data-centric XML abstraction in sorted, first-order predicate logic with equality. The formalization allows both the definition of constraints and updates. This unified representation in the same logic facilitates our main goal, to support the generation of weakest preconditions and to reduce them significantly. One main aspect of this reduction is the elimination of document manipulating operations, which is formulated in the central Theorem 3.2 of this section.

In the following, we present the essential sorts and symbols of the formalization and discuss the intended model. We only describe an informal semantics for the model here. Section 5 contains the important properties of non-standard symbols in the form of substitution rules. We assume that $\mathbb{Z}$ denotes the sort of integer numbers with constant, function and predicate symbols 0, 1, $+$, $-$, $*$ and $<$.

3.1 Documents and Paths

Our data-centric XML abstraction is centered around the concept of *paths*. We represent the structure of a document as a set of paths, which together make up a tree shape.

A path is either $\mathit{root} \in \mathbb{P}$ or a child $p/l[i] \in \mathbb{P}$ of another path p, where $l \in \mathbb{L}$ is a *label* and $i \in \mathbb{I}$ an *identifier*. The symbols *root* and $./.[.]$ are the constructors of $\mathbb{P}$. Some example paths, which are also valid considering the specification of Section 2, are the following:

```
root/inventory[null]/item[itemId]
root/inventory[null]/type[typeId]/size[null]
root/inventory[null]/capacity[null]
```

So a path is essentially a finite sequence of label/identifier pairs. In a sense, a label selects which *kind* of path is constructed, whereas the identifier selects which *instance* of that kind is addressed. In cases where we are not interested in differentiating instances, but want to select a default one, we use the special identifier *null*, which represents some fixed identifier of sort $\mathbb{I}$. We assume the models of $\mathbb{I}$ and $\mathbb{L}$ to be infinite sets and all constants of those sorts to be disjoint, i.e. the equality check on constants of $\mathbb{I}$ and $\mathbb{L}$ is trivial.

To simplify notion, *root* at the start of paths and the identifier *null* can be omitted. Note that this is different from XPath, where just giving a *nametest* selects all elements with that name. Thus, the above paths can be denoted as:

```
/inventory/item[itemId]
/inventory/type[typeId]/size
/inventory/capacity
```

In the example we could abbreviate paths even further, by using the implicit information inferable by the structural pattern. The double slash, for example, can be used to skip over trivial selection steps and the dot allows to refer to any path of the current kind, i.e. which has all identifiers universally quantified.

A set d of paths describes a tree if it fulfills two properties. It has to contain the *root* path and for each other path, it has to contain all prefix paths:

$$\mathit{root} \in d \qquad p/l[i] \in d \rightarrow p \in d$$

Using this path characterization of a tree, a *document* is defined by associating values to paths. Values could in general be of any datatype common to XML, like integers, floats, strings or identifiers, but for simplicity we restrict ourselves to integers and identifiers in this paper. Additionally, as identifiers are a central aspect of paths and hence for referencing in documents, we decided to assign both an integer value and an identifier to each path.

Consequently, we assume the model of sort $\mathbb{D}$ to be the finite maps from paths to integer/identifier pairs, yet we are only interested in those that satisfy the tree or *document property* defined above. In the following, we introduce notations for reading and manipulating documents.

3.1.1 Reading Documents

We define the $\in$ predicate on documents as the usual element inclusion on the domain of the document map, i.e. it is true iff a path is contained in a document.

There are two symbols used to read from documents, $d(p)_{\mathbb{Z}}$ for integers and $d(p)_{\mathbb{I}}$ for identifiers, which are the only other way beside $\in$ to observe a document d. Reading a path p from a document returns the according integer or identifier from the finite map. Reading from any path not included in a document is defined and yields some integer or identifier which does neither depend on the document nor the path. For all practical purposes we could, for example, assume default values ($0/\mathit{null}$) to be returned in these cases.

The *blank* document is the smallest possible document, i.e. it contains only the necessary *root* element:

$$p \in \mathit{blank} \Leftrightarrow p = \mathit{root}$$

$$\mathit{blank}(\mathit{root})_{\mathbb{Z}} = 0 \qquad \mathit{blank}(\mathit{root})_{\mathbb{I}} = \mathit{null}$$

As far as notation is concerned, we can neglect the sort annotation of $.(.)_{\mathbb{Z}}$ and $.(.)_{\mathbb{I}}$, as the context always makes it clear which one is necessary. As we are also neglecting the *root* at the start of paths, it is convenient to leave out parentheses as well:

```
$/inventory/capacity > 0
```

The example specification of Section 2 does not refer to any documents besides \$, so we most often skip that altogether.

As we now have all symbols necessary to define the second constraint of the example, we want to show it in our formalization. We also show how we can then abbreviate it depending on the context:

```
0 < $(root/inventory[null]/capacity[null])ℤ   (completely formal)
$/inventory/capacity > 0                      (schema independant)
//capacity > 0                           (by existence of a schema)
. > 0                                   (by location in the schema)
```

These abbreviations are vital to readability when expressing larger paths and when nesting paths to express references.

3.1.2 Manipulating Documents

There are only two operations necessary to make the transition from any starting document to any other document. The first operation $d[p \to (v, i)]$ adds a path/value pair to the map representing document d, possibly replacing an old pair associated with the path p. The second one $d[p \to]$ removes the pairs associated with path p, or any extension of p, from document d. We call these symbols *set* $.[. \to (.,.)]$ and *delete* $.[. \to]$, respectively.

3.2 Multisets

In order to support constraints on sums of values, or the number of children of an element, we need the ability to read more than single values. Such constraints are very common in the domain of tree-shaped data and are based on *aggregate* functions, which take multiple values at once. So in addition to the *unique* sorts we used so far, we also support the concept of *multisets*. In particular we are interested in the multiset sorts $\mathbb{P}^*$ (paths), $\mathbb{I}^*$ (identifiers) and $\mathbb{Z}^*$ (integers), which are marked by an additional star.

The model for these sorts is that of ordinary multisets, with the usual symbols like the multiset union $\cup$, the multiset difference $.\backslash.$, the subset relation $\subset$ or the empty multiset $\emptyset$. We will also need the $\times$ symbol, which repeats each member of a multiset. The unary symbol $\{.\}$ creates the singleton multiset. Instead of the normal multiset intersection $\cap$, we will need an intersection between a unique element and a set, which means intersecting the infinite repetition of the unique element with the multiset.

Additionally, to cope with $.[. \to]$ applications later on, we need the *deep intersection* $\Cap$ which is defined on $\mathbb{P} \times \mathbb{P}^*$ only. The result of such an intersection $p \Cap p^*$ is the subset of p^*, which includes only extensions of p.

The most important multisets are those of paths. With them it is possible to refer to all instances of a kind of paths at the same time. To be able to do so, we need more tools than the generic multiset connectives, so we define $./.[.]$ on path multisets, too.

The semantics of the child symbol regarding multiset parameters is that of a product. A path multiset created by $./.[.]$ therefore contains all paths which are any combination of a path from the parent path multiset, the given label and any identifier from the corresponding identifier multiset.

The correspondance to *root* on the multiset side is the singleton $\{\mathit{root}\}$. Even though $\{\mathit{root}\}$ and $./.[.]$ are not constructors for whole $\mathbb{P}^*$, they are still injective and have disjoint images.

To be able to create multisets of all instances of a kind of paths, we use the special multiset identifier *any*. This identifier represents complete $\mathbb{I}$ and using it in $./.[.]$ creates infinite path multisets.

3.2.1 Reading and Using Multisets

With $./.[.]$ defined on multisets, we also define the read operations $.(.)$ on multisets to create integer and identifier multisets. The predicate $\in$ is not defined on path multisets, as it is not needed as guard for $.(.)$ anymore. If a path is not in the document, no value is included for it in the resulting multiset.

So the result of a multiset read includes a value for each path which is both in the queried multiset and the document, which means the result can potentially be empty, but there is no unknown behavior. Note that as documents are finite maps, the result of a multiset read is also always finite.

One use of reading multisets from a document is to aggregate the values. We will work with the common functions *sum* and *count*, which sum up or count the values in an integer multiset.

Identifier multisets are most often used in $./.[.]$ applications, to reference a whole multiset of elements. Where *any* selects all instances of a kind, using an identifier multiset read from the document creates a referenced multiset. A common example is the sum of the multiset of salaries of the employees of a company.

```
sum($//salary[$//employee*/salary])
```

We are using the *any* identifier multiset to select all employees and use the resulting identifier multiset to reference salaries, which are then summed up. It is convenient to use a simple $*$ instead of $[\mathit{any}]$. Note that in contrast to XPath, selecting more than one element is explicit in our notation.

3.3 Assertions

The ultimate goal of our formalization is to formulate assertions on documents, express the semantics of procedures and express weakest preconditions. With all necessary sorts and symbols introduced, we now formally define *assertions.*

Definition 1. The sorted signature σ_0 contains the sorts

$$S = \{\mathbb{Z}, \mathbb{Z}^*, \mathbb{I}, \mathbb{I}^*, \mathbb{P}, \mathbb{P}^*, \mathbb{D}\}$$

as well as the predicate and function symbols:

$<$	$:: \mathbb{Z} \times \mathbb{Z}$	$\in$	$:: \mathbb{P} \times \mathbb{D}$
$=$	$:: a \times a$		
$0, 1$	$:: \mathbb{Z}$	$+, -, *$	$:: \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$
$.(.)_\mathbb{Z}$	$:: \mathbb{D} \times \mathbb{P} \rightarrow \mathbb{Z}$	$.(.)_\mathbb{Z}$	$:: \mathbb{D} \times \mathbb{P}^* \rightarrow \mathbb{Z}^*$
count	$:: \mathbb{Z}^* \rightarrow \mathbb{Z}$	*rcount*	$:: \mathbb{I} \times \mathbb{I}^* \rightarrow \mathbb{Z}$
sum	$:: \mathbb{Z}^* \rightarrow \mathbb{Z}$		
null	$:: \mathbb{I}$	*any*	$:: \mathbb{I}^*$
$.(.)_\mathbb{I}$	$:: \mathbb{D} \times \mathbb{P} \rightarrow \mathbb{I}$	$.(.)_\mathbb{I}$	$:: \mathbb{D} \times \mathbb{P}^* \rightarrow \mathbb{I}^*$
root	$:: \mathbb{P}$		
$./.[.]$	$:: \mathbb{P} \times \mathbb{L} \times \mathbb{I} \rightarrow \mathbb{P}$	$./.[.]$	$:: \mathbb{P}^* \times \mathbb{L} \times \mathbb{I}^* \rightarrow \mathbb{P}^*$
blank	$:: \mathbb{D}$		
$.[.\rightarrow]$	$:: \mathbb{D} \times \mathbb{P} \rightarrow \mathbb{D}$	$.[.\rightarrow(.,.)]$	$:: \mathbb{D} \times \mathbb{P} \times \mathbb{Z} \times \mathbb{I} \rightarrow \mathbb{D}$
$\emptyset$	$:: a^*$	$\cup$	$:: a^* \times a^* \rightarrow a^*$
$\{.\}$	$:: a \rightarrow a^*$	$\times$	$:: \mathbb{Z} \times a^* \rightarrow a^*$

The signature σ, which contains all symbols of the formalization, is defined as signature σ_0, with the additional *helper* symbols:

$\cap$	$:: a \times a^* \rightarrow a^*$	$\subset$	$:: a^* \times a^*$
$\Cap$	$:: \mathbb{P} \times \mathbb{P}^* \rightarrow \mathbb{P}^*$	$.\backslash.$	$:: a^* \times a^* \rightarrow a^*$
$\rightsquigarrow$	$:: \mathbb{P} \times \mathbb{P}$		

Definition 2. An *assertion* is a first-order formula over the signature σ_0

- with equality restricted to $\mathbb{Z}$ (integers) and $\mathbb{I}$ (identifiers),
- which contains only variables of sorts $\mathbb{Z}$ and $\mathbb{I}$, as well as the special variable \$ of sort $\mathbb{D}$ (documents),
- with quantification restricted to universal quantification of variables of sort $\mathbb{I}$ at the top level.

A *constraint* is an assertion without free variables except \$ and no document manipulating operations, i.e. neither $.[.\rightarrow(.,.)]$ nor $.[.\rightarrow]$.

Definition 3. A *restricted term* is a term (of any sort) over signature σ_0

- which contains only variables on $\mathbb{Z}$ and $\mathbb{I}$, as well as the special variable \$ of sort $\mathbb{D}$.

These are the terms that can appear in an assertion.

LEMMA 3.1. *Given any assertion and free variable assignment, including a document bound to \$, the assertion can be evaluated to true or false.*

PROOF. All unquantified variables and expressions can be evaluated by using the variable assignment. An assertion can only contain universal quantifiers on variables of sort $\mathbb{I}$. As documents are finite, however, all predicates with paths containing quantified identifiers can be evaluated by finite case analysis. Note that all identifiers which create a path not included in the document can be combined to a single case. $\square$

We conclude this section with our main theorem. It will be necessary for precondition checking in Section 4, where it helps to significantly reduce their size. Its proof is based on numerous simplification rules presented in Section 5.

THEOREM 3.2. *Any assertion can be normalized into an equivalent assertion not containing operations for document manipulation.*

PROOF. There are only three ways to observe a document and two different manipulating operations. Lemma 5.1, 5.2 and 5.3 cover all possible cases and can together unconditionally normalize an assertion to no longer contain document manipulating operations. $\square$

4. PROCEDURES AND INVARIANTS

In order to use the two manipulation operations of the formalization, we define a core procedural language around them. Our goal is to show how we can handle local manipulations and especially reason about their weakest precondition regarding global invariants.

4.1 Core Language

Definition 4. A *procedure* is a sequence of basic operations, with arbitrary many parameters of sorts $\mathbb{Z}$ and $\mathbb{I}$, represented by free variables in terms. A *basic operation* is one of the following, where paths p, integers v and identifiers i are *restricted* (Def. 3):

- **app** p v - Appends the path p, assigning the value v (and the default identifier *null*). The parent of p has to exist in the current document and the appended element must not already exist.
- **upd** p v - Updates the integer of an existing path p, with v (leaving the identifier as it is).
- **rem** p - Removes the unique, existing path p, together with all descendant paths, from the document.
- **link** p i - Sets the identifier of an existing path p to i (leaving the integer as it is).

The operations diverge if they do not meet their requirements.

In the example of Section 2 we used a higher level language and especially a more convenient syntax to append multiple paths, by giving an XML fragment with embedded value terms. In the core language, the procedure `addItem` is expressed by the following sequence of operations:

```
app   /inventory/item[itemId]          0
app   /inventory/item[itemId]/typeref  0
app   /inventory/item[itemId]/since    $/inventory/time
link  /inventory/item[itemId]/typeref  typeId
```

The semantics of the core language is given in an axiomatic way by Hoare-triples. We assume the usual Hoare rules, in particular for sequencing, and focus on the axioms for the basic operations. The currently manipulated document is represented by the special program variable \$, which is the only variable modified in each Hoare-triple. All other variables, especially the value D of the document in the prestate, remain the same:[1]

$$\begin{array}{lll}
\left\{p \in \$ \wedge p/l[i] \notin \$ \wedge \$ = D\right\} & \textbf{app}\ p/l[i]\ v & \left\{\$ = D[p/l[i] \rightarrow (v, \mathit{null})]\right\} \\
\left\{p \in \$ \wedge p \neq \mathit{root} \wedge \$ = D\right\} & \textbf{upd}\ p\ v & \left\{\$ = D[p \rightarrow (v, D(p)_{\mathbb{I}})]\right\} \\
\left\{p \in \$ \wedge p \neq \mathit{root} \wedge \$ = D\right\} & \textbf{rem}\ p & \left\{\$ = D[p \rightarrow]\right\} \\
\left\{p \in \$ \wedge p \neq \mathit{root} \wedge \$ = D\right\} & \textbf{link}\ p\ i & \left\{\$ = D[p \rightarrow (D(p)_{\mathbb{Z}}, i)]\right\}
\end{array}$$

THEOREM 4.1. *Any non-diverging execution of a procedure does not violate the document property of* \$.

PROOF. The *root* cannot be removed and its default values cannot be changed. The `append` operation explicitly checks for the existence of the parent path. The `update` and `link` operations only replace an existing mapping, i.e. they do not alter the domain of the document mapping at all. Finally, the `remove` operation removes a path together with all descendants, so the prefix property cannot be violated. □

4.2 Weakest Preconditions

The weakest preconditions for the basic operations, which are so closely related to the formalization, are very similar to the definition of their semantics. The requirements for successful execution remain exactly the same. For the constraint C to hold in the poststate, however, it needs to hold in the prestate, where all occurrences of the current document are replaced with the manipulated one. This part of each triple is exactly the same as in the assignment axiom of Hoare-Logic:[2]

$$\begin{array}{lll}
\left\{p \in \$ \wedge p/l[i] \notin \$ \wedge C_{\left[\$[p/l[i] \rightarrow (v, \mathit{null})]/\$\right]}\right\} & \textbf{app}\ p/l[i]\ v & \left\{C\right\} \\
\left\{p \in \$ \wedge p \neq \mathit{root} \wedge C_{\left[\$[p \rightarrow (v, \$(p)_{\mathbb{I}})]/\$\right]}\right\} & \textbf{upd}\ p\ v & \left\{C\right\} \\
\left\{p \in \$ \wedge p \neq \mathit{root} \wedge C_{\left[\$[p \rightarrow]/\$\right]}\right\} & \textbf{rem}\ p & \left\{C\right\} \\
\left\{p \in \$ \wedge p \neq \mathit{root} \wedge C_{\left[\$[p \rightarrow (\$(p)_{\mathbb{Z}}, i)]/\$\right]}\right\} & \textbf{link}\ p\ i & \left\{C\right\}
\end{array}$$

We use the notation $C_{[b/a]}$ to refer to the formula C, where all occurrences of a are substituted with b. We assume that this substitution is collision-free, i.e. that all variables bound to a quantifier in C, which are used as free variables in an argument p, v or i, are renamed. To refer to the weakest precondition of a given formula C and procedure P, we use the standard notation $wp(P, C)$.

THEOREM 4.2. *The weakest precondition of an* assertion *C, regarding a procedure, can be normalized into an* assertion.

PROOF. As the parameters of basic operations are restricted terms, the substitutions on C do not violate the assertion property. The additional formulas introduced in preconditions are existence checks and equalities on paths. The former pose no violation to being an assertion, whereas the latter can be normalized by Lemma 5.4, in this special case even to *true* or *false*.

Finally, The universal quantifiers of C can be pulled to the top level of the precondition by using standard first-order theorems on conjunctions. □

[1]Note that by being restricted, the paths in operations are either *root* or a ./.[.] application. Append diverges for *root*, so the triple covers only the other case, and we are able to express the necessary condition on the parent this way.

[2]The weakest precondition of appending the *root* is obviously *false*.

4.3 Invariants

With both the assertion language and the core procedural language defined, we now formulate what invariants are and how they are maintained.

Definition 5. An *invariant* is a set of one or more *constraints*. We call a document *valid* with regard to an invariant, if all constraints evaluate to *true*, when binding the document to the free variable \$.

Without loss of generality we can assume an invariant to be a single constraint, as multiple constraints can be combined in a conjunction and their quantifiers be pulled out to get a constraint again. It is a useful compositional argument, however, that invariants most often split into fairly independent parts, for which the weakest precondition generation, as well as large parts of the simplification, can be done in parallel.

In Section 2 we used a higher level language to conveniently define various kinds of constraints. The following listing shows the resulting invariant as collection of constraints, in a schema-independent[3], but still abbreviated notation:

```
Structural Constraints:

/inventory ∈ $
/inventory/time ∈ $
/inventory/capacity ∈ $

∀x∈𝕀: /inventory/type[x] ∈ $ → /inventory/type[x]/size ∈ $
∀x∈𝕀: /inventory/type[x]/size ∈ $ → /inventory/type[x] ∈ $

∀x∈𝕀: /inventory/item[x] ∈ $ → /inventory/item[x]/since ∈ $
∀x∈𝕀: /inventory/item[x] ∈ $ → /inventory/item[x]/typeref ∈ $
∀x∈𝕀: /inventory/item[x]/since ∈ $ → /inventory/item[x] ∈ $
∀x∈𝕀: /inventory/item[x]/typeref ∈ $ → /inventory/item[x] ∈ $

Integrity Constraints:

∀x∈𝕀: /inventory/item[x]/typeref ∈ $ →
    /inventory/type[$/inventory/item[x]/typeref] ∈ $

Value-Based Constraints:

$/inventory/capacity > 0
$/inventory/time ≥ 0

∀x∈𝕀: /inventory/type[x]/size ∈ $ →
    $/inventory/type[x]/size > 0
∀x∈𝕀: /inventory/type[x]/size ∈ $ →
    $/inventory/type[x]/size ≤ $/inventory/capacity

∀x∈𝕀: /inventory/item[x]/since ∈ $ →
    $/inventory/item[x]/since ≥ 0
∀x∈𝕀: /inventory/item[x]/since ∈ $ →
    $/inventory/item[x]/since ≤ $/inventory/time

$/inventory/capacity ≥
    sum ($/inventory/type[$/inventory/item*/typeref]/size)
```

Note that we would normally neglect the universal quantifiers in the listing, but their explicit presence emphasizes the difference between unique constraints and those which are instantiated for each existing path.

We described the technique to maintain invariants in Section 2.3. Our formalization is designed in such a way that a nearly optimal[4] C_s can often be found fully automated.

[3]The notation is schema-independent in that no path is relative or abbreviates trivial child steps. The schema, however, is completely represented within the invariant.

[4]We consider optimality in terms of runtime efforts, not textual length, although these criteria often coincide.

Starting with an invariant C, Theo. 4.2 guarantees $wp(P, C)$ to trivially normalize to an assertion. Lemma 3.1 shows us that $wp(P, C)$ can be checked on the initial document. By Theorem 3.2 it can be normalized to resemble the original invariant C in large parts, by removing all document manipulating operations.

Section 5 introduces all the necessary rules to do so and also shows in which ways the resulting, normalized assertion is altered. We will exploit this fact in Section 6, where we argue that our implementation comes indeed close to finding an optimal C_s.

5. RULES

Weakest precondition generation for invariants produces large formulas containing document manipulations. In this section we define the necessary rules to remove document manipulating operations from assertions. By doing so we are able to automatically derive a precondition which can be checked on the initial document and do not require the programmer to give a specification. Using the rules of this section, a precondition can be drastically simplified, up to a form where it contains only the really necessary checks. Minimality is important, as the precondition has often to be checked at runtime.

The rules we present are directly based on the semantics of $\in$ and the unique and multiset read operations. Using these rules, we can formulate the three Lemmata 5.1, 5.2 and 5.3, which allow us to prove the essential Theorem 3.2.

Definition 6. A *substitution rule* has the form

$$\frac{A}{B}\ F$$

and states that any matching term or formula A can be replaced with B, whenever condition F is established. We call a rule *correct*, if applying it yields an equivalent ($\Leftrightarrow$) formula or equal ($=$) term, respectively.

So every rule is based on a theorem of the form $F \rightarrow A \Leftrightarrow B$ for formulas or $F \rightarrow A = B$ for terms, where F can just be *true*. In many cases we will present pairs of rules, which taken together can unconditionally simplify a term or formula. A simple example for a pair of rules on integers with multiplication is the following:

$$\frac{x * a = x * b}{a = b}\ x \neq 0 \qquad \frac{x * a = x * b}{\mathit{true}}\ x = 0$$

Using this pair we can simplify a formula of the form $x * a = x * b$ unconditionally to $(x = 0 \rightarrow \mathit{true}) \wedge (x \neq 0 \rightarrow a = b)$ or simply $(x = 0) \vee (a = b)$. A pair of rules on terms can be used in the same way, but the case analysis $(F \rightarrow P_1) \wedge (\neg F \rightarrow P_2)$ has to be done on the enclosing predicate P.

5.1 Domain Check Elimination

A path for which values were recently set is always in the document, no matter if it was before. Setting values for any path different from the queried one is irrelevant, i.e. its existence depends on the unmodified document.

$$\frac{p' \in d[p \rightarrow (v, i)]}{\mathit{true}}\ p = p' \qquad \frac{p' \in d[p \rightarrow (v, i)]}{p' \in d}\ p \neq p'$$

A path is not in the document, if it has been deleted from the document. This can either have happened directly or because one of its ancestor paths has been deleted. Likewise a deletion is irrelevant if the queried path is not a descendant of the deleted one.

$$\frac{p' \in d[p \rightarrow]}{\mathit{false}}\ p \rightsquigarrow p' \qquad \frac{p' \in d[p \rightarrow]}{p' \in d}\ p \not\rightsquigarrow p'$$

These rule pairs can both be used by direct case analysis on the $\in$ predicate, to eliminate $.[. \rightarrow (.,.)]$ and $.[. \rightarrow]$ applications from an assertion. Note that these rules introduce equality on paths, which is forbidden in an assertion, and the ancestor predicate $\rightsquigarrow$. This predicate is one of the helper symbols introduced to assertions by substitution rules. We will show how to deal with these symbols in Section 5.4.

LEMMA 5.1. *Any assertion can be normalized into an equivalent assertion not containing document manipulating operations in $\in$ applications.*

PROOF. We presented two pairs of rules, which together allow to unconditionally remove $.[. \rightarrow (.,.)]$ and $.[. \rightarrow]$ from $\in$ applications. By applying these rule pairs, we are introducing both equality on restricted paths and $\rightsquigarrow$ on restricted paths into the assertion. The Lemmata 5.4 and 5.5 show how both can be normalized to identifier equalities. $\square$

5.2 Unique Read Elimination

Reading from a path for which values were recently set returns the corresponding value:

$$\frac{d[p \rightarrow (v, i)](p')_{\mathbb{Z}}}{v}\ p = p' \qquad \frac{d[p \rightarrow (v, i)](p')_{\mathbb{I}}}{i}\ p = p'$$

Setting the values of any other path does not have an effect. Likewise deleting paths which are not ancestors of the queried path has no effect, so we can read from the unmodified document:[5]

$$\frac{d[p \rightarrow (v, i)](p')}{d(p')}\ p \neq p'$$

$$\frac{d[p \rightarrow](p')}{d(p')}\ p \not\rightsquigarrow p' \qquad \frac{d[p \rightarrow](p')}{\mathit{blank}(p')}\ p \rightsquigarrow p'$$

Reading from a path which is not contained in a document does not depend on the document, so we can as well read from the blank document. Note that these rules again introduce path equality and ancestor checks, which have to be removed for the resulting formula to be an assertion.

LEMMA 5.2. *Any assertion can be normalized into an equivalent assertion not containing document manipulating operations in unique $.(.)_{\mathbb{Z}}$ and $.(.)_{\mathbb{I}}$ applications.*

PROOF. We presented four pairs of rules, which together allow to unconditionally remove $.[. \rightarrow (.,.)]$ and $.[. \rightarrow]$ from $.(.)$ applications (two pairs for $.(.)_{\mathbb{Z}}$ and two for $.(.)_{\mathbb{I}}$). By applying these rule pairs, we are introducing both equality on restricted paths and $\rightsquigarrow$ on restricted paths into the assertion. The Lemmata 5.4 and 5.5 show how both can be normalized to identifier equalities only.

Note that these identifiers might as well be $.(.)_{\mathbb{I}}$ applications on manipulated documents, which can also be normalized by this lemma. The normalization is guaranteed to terminate, as the nesting of $.(.)$ applications is finite. $\square$

[5]As these rules do not differ for $.(.)_{\mathbb{Z}}$ and $.(.)_{\mathbb{I}}$, we define them together by neglecting the sort annotation.

5.3 Multiset Read Elimination

Reading a multiset which may include a path recently modified by $.[.\rightarrow(.,.)]$ is just reading the former multiset, removing all manipulated values and adding the corresponding new values of the manipulated elements:

$$\frac{d[p \rightarrow (v,i)](p^*)}{d(p^*)\backslash d(p \cap p^*) \cup d[p \rightarrow (v,i)](p \cap p^*)}$$

Note that it is irrelevant if p was added to the document or just updated, as $d(p \cap p^*)$ will be the empty set or the set of old values accordingly. Additionally we do not need to differentiate between p being included in p^* or not, as the intersection will just be empty in the former case and represent the modified elements in the latter.

Reading a multiset with one or more paths removed is straight-forward the multiset of old values with the deleted ones removed.

$$\frac{d[p \rightarrow](p^*)}{d(p^*)\backslash d(p \Cap p^*)}$$

Again it is not necessary to make any direct case analysis, as the deep intersection semantics correctly covers all possible cases.

Note that the rules do not need any direct case analysis, yet duplicate the read operation. They also introduce multiset intersection $\cap$, multiset difference $.\backslash.$ and the deep intersection $\Cap$.

Lemma 5.3. *Any assertion can be normalized into an equivalent assertion not containing document manipulating operations in multiset $.(.)_\mathbb{Z}$ and $.(.)_\mathbb{I}$ applications.*

Proof. The proof of this Lemma is too complex for the scope of this this paper and is based on several additional Lemmata which cannot be included due to space limitations. The basic idea is to reduce parameters to straight paths and apply the two rules shown above, which then introduce multiset differences and both normal and deep intersections on straight paths. All these can be removed, however, as the multiset difference is always on subsets and intersections can be reduced to unique or repeated unique paths. $\Box$

5.4 Helper Symbol Elimination

The main rules of this section all introduce symbols which must not normally appear in assertions and consequently have to be normalized. These helper symbols can be removed with additional rules.

5.4.1 Path Equality

By restricting the sorts of variables, paths in assertions are *restricted* and become explicit. A path is always absolute, starts with *root*, and does explicit $./.[.]$ steps using constant labels. Only the used identifiers can be variable, which make the path generic. As a consequence, any path equality introduced to an *assertion* by using a *rule* can be statically decided up to identifier equality.

Lemma 5.4. *Any formula containing equality on restricted paths can be normalized into an equivalent formula not containing such equalities. In particular, path equality on restricted paths reduces to identifier equalities only.*

Proof. A restricted path is either *root* or a $./.[.]$ application. The three resulting cases can be covered by recursively applying one of the three rules:

$$\frac{root = root}{true} \qquad \frac{root = p/l[i]}{false} \qquad \frac{p_1/l_1[i_1] = p_2/l_2[i_2]}{p_1 = p_2 \wedge l_1 = l_2 \wedge i_1 = i_2}$$

The label equality of the third rule reduces trivially to *true* or *false*. The normalization terminates as paths are finite and they are either eliminated or reduced in length by each rule application. $\Box$

5.4.2 Ancestor

The $\rightsquigarrow$ predicate defines the ancestor relationship on paths. Basically, a path p_1 is an ancestor of p_2, if there is a sequence of $./.[.]$ applications, which leads to p_2 when applied to p_1.

As with path equality, any ancestor predicate introduced to an *assertion* by using a *rule* can be statically decided up to identifier equality.

Lemma 5.5. *Any formula containing $\rightsquigarrow$ applications on restricted paths can be normalized into an equivalent formula not containing such applications. In particular, $\rightsquigarrow$ on restricted paths reduces to identifier equalities only.*

Proof. The rules we are using to normalize $\rightsquigarrow$ are based on reducing the right hand path to *root*. As this is a restricted path, there are only two cases and hence two rules:

$$\frac{p \rightsquigarrow root}{p = root} \qquad \frac{p \rightsquigarrow p'/l[i]}{p = p'/l[i] \vee p \rightsquigarrow p'}$$

Only *root* itself is an ancestor of *root*. In all other cases, a path is an ancestor of another iff it is either equal to the other path or an ancestor of its parent. As paths are finite, this guarantees the normalization to terminate. The path equalities introduced are always on restricted paths and can be normalized to identifier equalities by Lemma 5.4. $\Box$

5.4.3 Multisets

Eliminating multiset intersections and differences is a complex task and needs more technical detail than necessary for this paper. Due to space constraints we cannot include those Lemmata in this work.

6. IMPLEMENTATION

We developed a specification language for XML formats as well as manipulating procedures and have implemented compilers targeting the formalization proposed in this paper. We also implemented tool support for the formalization. It is possible to derive the weakest precondition of procedures, simplify them and further reduce them using the invariant.

Of all this, the biggest complexity lies with the simplifier. Eliminating document manipulating operations can blow up the assertion, which has to be reduced again. Assertions affected by substitution rules can often be simplified further or be eliminated altogether. In this section we describe techniques used and the overall complexity of a resulting average precondition.

6.1 Normalization

In order to deal with assertions effectively, we are using the conjunctive normal form (CNF). A formula in CNF is a conjunction of disjunctions of predicates, which can be negated. Additionally the formula can be universally quantified at the top level.

This form is very natural to the domain of tree-shaped data and XML in general. It directly reflects a rule-based approach to specification, in that each conjunct is a rule and each of its disjuncts a case of the rule which can make it valid. Adding additional rules to an invariant is just adding conjuncts.

The usual problem with CNF is that an arbitrary first-order formula is only guaranteed to normalize to an *equisatisfiable* formula in CNF. This is due to the necessary skolemization of existential quantifiers. But as *assertions* do not contain existential quantifiers, and universal ones can only appear at top level, we know that every assertion can be normalized to an *equivalent* assertion in CNF.

6.2 Case Analysis Complexity

In order to eliminate manipulations from an assertion, it is necessary to do case analysis in formulas, which effectively duplicates the whole formula. In most cases, however, one of the two cases can then immediately be simplified to a trivial formula, often even *true* or *false*.

Nevertheless this duplication and following simplification creates extra efforts, most of which can be circumvented right from the start. The reason for case analysis to be necessary is to decide path equality for $.(.)$ and $\in$ applications. So basically it would be sufficient to duplicate these predicates only, instead of the whole formula.

As we are using CNF, however, and as rules very often contain multiple similar paths anyway, we judge it most efficient to duplicate the conjunct which contains the predicate in question. This keeps the constraint in CNF and duplicates only a very small amount of an invariant, while anticipating other equal case analysis steps in $.(.)$ and $\in$ applications in the same conjunct.

When looking at the manipulation elimination rules used in Lemma 5.1, 5.2 and 5.3, we get an insight on the kind of substitutions done. For domain checks and unique reads we use rule pairs where the one rule just removes the manipulation, while the other is greatly simplified and does no longer contain the document at all.

The rules for multiset read do not do any case analysis directly, but multiply the original read operation. This is a simple necessity, not a shortcoming of the formalization. Especially when using paths that contain nested identifier multiset reads for referencing, the resulting preconditions are bound to be complex. Nevertheless, the two rules also trigger case analysis, as they introduce intersections that have to be normalized. One of the cases, however, always results in the intersection being the empty set, so the term collapses to the original read without the document manipulation.

As a consequence, when doing case analysis on conjuncts, we get one conjunct which is a weakened form of the original one, while the other is adapted to the nature of manipulation and represents the necessary check on parameter values and the initial document. Our normalization splits up and isolates the relevant parts of quantified constraints. The remaining common part can be eliminated by using the invariant.

This is why a constraint which does not contain any manipulated path will never increase the size of the precondition. There will be cases in which the adjusted part of a split up constraint still contains other quantifiers. This happens for constraints on a quantified path, when an also referenced ancestor is manipulated. This is not a shortcoming of our formalization, but an inherently complex situation.

Consider a procedure updating the global time of our example in Section 2. This single manipulation is relevant for *all* `since` attributes, of every `item` in the inventory. The reference to $//time$ in the constraint is substituted by whatever value the procedure sets, so the constraint is not necessarily implied by the invariant and consequently has to be checked. If we simply increment `time` by one, however, the resulting precondition is a weakened form of the original and can be dropped by the simplifier.

6.3 Simplification

After isolating the relevant parts of the original invariant, and eliminating the remaining common ones, the precondition can still be severely simplified. Additionally, weakest precondition generation does not only add document manipulating operations to the invariant, but also domain checks and a trivial equality. The domain checks represent the necessary precondition to maintain the document property of \$.

While the presence of a structural schema is not necessary for our approach, it greatly helps in reducing the precondition. Like in our example specification, the existence of many paths is bound to that of others. By using these implications, domain checks can be reduced to the absolutely necessary minimum, like also shown in the example.

When a constraint gets adjusted by a rule, it often happens that it becomes a weakened form of another one or a tautology, so it can be dropped. This often is the implementation idea the programmer had in mind, like with the *since* attribute in the example. Setting this attribute to $//time$ splits up two adjusted constraints for the `since` attribute of the `item` with identifier `itemId`:

```
$//time ≥ 0
$//time ≤ $//time
```

The first is exactly the constraint on the `time` attribute itself, while the second is just a trivial tautology.

All these simplifications can essentially be done by using only two rules:

$$\frac{(l_1 \vee ... \vee l_n) \wedge (l_1 \vee ... \vee l_n \vee ... \vee l_m)}{l_1 \vee ... \vee l_n} \qquad \frac{a \wedge (\neg a \vee b)}{a \wedge b}$$

The first is the absorption rule, which gets rid of weakened conjuncts. The second is an adaption of modus ponens, to eliminate guards of implications, if they have already been established. This rule also deals with inferring structural properties from the invariant, by instantiating quantified implications with the concrete identifiers necessary.

6.4 Experiences

Using our prototype implementation, we have specified schemata, as well as atomic manipulation procedures, for some example systems. The schema of the largest example, which describes a management system for Bachelor students and the official module handbook, is specified in about 90 lines[6] in the style of the example in Sect. 2.

This specification can be read by the system and be normalized to a single CNF formula. In the following we are using the size of the formula as metric, where each predicate and term constructor counts as one. The example then

[6]Not counting comments and empty lines.

amounts for 186 constraints, with a combined complexity of about 6.000.

One of the most basic procedures of the example system is `addStudent`, which consists of 10 simple append operations. Generating the weakest precondition of this procedure, and applying trivial simplification rules, still produces 20 more constraints (two for each append) and blows up the invariant to a complexity of over 60.000.

Our simplifier can immediately reduce this precondition back to 195 constraints, measuring only 6.800, just by doing basic simplifications and eliminating meaningless document manipulating operations (without case analysis). After that, another step removes constraints which are just unmodified parts of the invariant and we are down to 46 constraints of about 1.800 complexity.

The remaining constraints are analyzed by case analysis, where one case is often *true* and the other a weakened part of the invariant, so it is dropped. This analysis is also fully automated and done by the simplifier, which is now down to a **single** constraint of complexity 10. This constraint states that the added student must not already exist in the system (`/uni/students/student[sid]` $\not\in$ `$`). This matches our initial feeling that adding a new student to the system should not violate any constraints at all. Note that the precondition of the example in Sect. 2 was also produced with our implementation.

7. RELATED WORK

Similar to Context Logic [8], we have the goal to prove properties about tree-manipulating procedures. Calcagno, et al. are interested in local reasoning about manipulating procedures, by abstracting the context of a subtree. This is done in the spirit of separation logic, by introducing an asymmetric spatial conjunction that captures arbitrary tree contexts. They are presenting a technique to automatically derive weakest preconditions of procedures, which can be used to prove complex procedures correct w.r.t. their specifications.

While these techniques might work well for verification challenges and to prove correctness of pre-/postcondition pairs of procedures, they are not well suited to specify constraints over data. Formulas in context logic are not easy to read for common programmers and the locality of constraints is expressed in the structure of the formula, rather than by giving a localizing path. It is unclear how XML Schemata can be expressed and the concept of paths is not supported at all. For our work, we assume a setting where procedures do not have a specification, but are required to maintain the validity of data without knowing the other clients. We therefore do not prove procedures correct, but we need to dynamically check the necessary parts of preconditions at runtime. It is unclear how complex checking weakest preconditions in context logic is, especially for XML Schemata which have to be maintained.

We are taking a different approach in that we are capturing locality by structuring a tree using labels. While such labels exist in context logic, they are not used to identify nodes. This is done by using globally unique identifiers – a heritage from the heap model of separation logic. With our approach identifiers are only locally unique and nodes of a tree are identified by labelled paths, much like it is done in Storeless Semantics [6]. We are also structuring the heap in a tree of subheaps. By restricting basic manipulating operations to specific subheaps, we achieve the locality necessary to isolate affected parts of an invariant.

Additionally, both works focus on the programming languages side, targeting expressive languages and complex algorithms. We are interested in maintaing properties of data structures, which are manipulated with basic, atomic procedures. These procedures can then be used by higher level programming languages to implement any kind of algorithm. Our goal is to enable domain experts, which most often do not have a background in verification, to write both declarative constraint specifications and simple procedures, for which they do not need to write a specification at all.

Bouchou and Alves [5] are also interested in maintaining validity of documents in the presence of updates and especially focus on doing so incrementally. The main difference to our work is that they regard structural validity only, but support more complex constraints for that than we do. In contrast, we also consider integrity constraints and especially value-based constraints on top of structure, which we haven't seen yet in the literature at all.

Benedikt and Koch describe and analyze different subsets of the XPath language in their survey [21]. The classification introduced there is interesting, as it matches our observations while searching for a suitable subset for this work. Basically, we use a subset of *aggregate XPath*, which exactly supports the functions `sum` and `count`, as well as binary operators on integer, but we severely restrict filter expressions, as they have to be controlled carefully to allow static analysis. We also identified a subset of *navigational XPath* in the sorts $\mathbb{P}$ and $\mathbb{P}^*$.

Much work has been done on related topics like integrity constraints, path constraints, analysis of XPath subsets (with or without structural constraints), etc. [2, 3, 7, 13, 14, 15, 16, 17]. However, the central focus of these papers is on expressibility, decidability and further properties like e.g. the constraint consistency and subsumption problems. In comparison to our specification language, several of these papers also cover XPath features that we do not yet support like e.g. navigational axes and structural filters. On the other hand, we can handle value-based constraints which are of practical importance w.r.t. our goals. In the future, we aim to extend our specification language and hope to exploit formalization and proof techniques of the cited papers.

Specialized logics for trees and local reasoning [4, 8, 9, 20, 22] also focus on expressibility and feature-completeness, where arbitrary trees and properties about trees can be represented. For our purposes these are too general and complex, which immediately renders automated methods for our goals unnecessary complicated. Nevertheless those ideas and techniques might prove useful for future work.

As far as local update languages are concerned, we did not find a suitable candidate. This led us to defining a procedural core language based on XPath, which is somewhat similar in expressibility to a subset of the discontinued XUpdate [19] project. James Cheney made the same observation in [10] and proposed a lightweight XML update language called LUX, which is the most interesting candidate for a more powerful update language to support in future work. He also refers to a lot of existing languages to manipulate XML, which either follow a different paradigm, e.g. document transformation, or are too complex for the needs of our work.

Cheney uses Hoare reasoning for static type checking of a type system in the spirit of regular expression types. Such type systems are extremely flexible considering the structure of documents, but neither catch integrity constraints nor value based constraints. Their analysis is completely static and boils down to programs either being correct or incorrect, without the need to create preconditions to check at runtime.

8. CONCLUSION

In this paper, we have laid the formal foundations for automated methods to maintain invariants on tree-shaped data, especially XML. We have formalized a data-centric XML abstraction in first-order logic, which is carefully designed to facilitate this goal. Paths are a central concept of this formalization, as they allow to precisely identify elements, carry structural information and allow to use the structure of a document in formal reasoning on constraints.

Our presented formalization supports structural and integrity constraints, but also value-based constraints, and additionally captures the semantics of local manipulation languages. This combination not only allows us to reason about procedures and derive their weakest precondition, but we are able to automatically isolate affected parts of an invariant and eliminate the others. As a result, invariants on data-centric XML can be maintained by checking only the relevant constraints on the current document and parameter values.

On top of these foundations, we have developed a higher level specification language which we used for the example and discussed practical experiences in Section 6. These evaluations show that our approach is valid and point to the direction of future work. We aim to incorporate more features into the specification language in a way that they can be handled by our formalization technique.

9. REFERENCES

[1] *PLAN-X 2007, Programming Language Technologies for XML, An ACM SIGPLAN Workshop colocated with POPL 2007, Nice, France, January 20, 2007*.

[2] S. Abiteboul and V. Vianu. Regular path queries with constraints. In *PODS '97*, pages 122–133, New York, NY, USA, 1997. ACM.

[3] M. Benedikt, W. Fan, and F. Geerts. Xpath satisfiability in the presence of DTDs. In *PODS '05*, pages 25–36, New York, NY, USA, 2005. ACM Press.

[4] N. Biri and D. Galmiche. Models and separation logics for resource trees. *J. Log. and Comput.*, 17(4):687–726, 2007.

[5] B. Bouchou, M. Halfeld, and F. Alves. Updates and incremental validation of XML documents. In *In 9th International Workshop on Database Programming Languages*, pages 216–232. ACM Press, 2003.

[6] M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *PEPM '03*, pages 55–65, New York, NY, USA, 2003. ACM.

[7] P. Buneman, W. Fan, and S. Weinstein. Path constraints on semistructured and structured data. In *PODS '98*, pages 129–138, New York, NY, USA, 1998. ACM.

[8] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. *SIGPLAN Not.*, 40(1):271–282, 2005.

[9] L. Cardelli and A. D. Gordon. Anytime, anywhere: modal logics for mobile ambients. In *POPL '00*, pages 365–377, New York, NY, USA, 2000. ACM.

[10] J. Cheney. Lux: A lightweight, statically typed XML update language. In *PLAN-X* [1], pages 25–36.

[11] J. Clark. RELAX NG compact syntax, Nov. 2002. `http://www.oasis-open.org/committees/relax-ng/compact-20021121.html`.

[12] J. Clark and M. Makoto. RELAX NG specification, Dec. 2001. `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`.

[13] A. Deutsch and V. Tannen. Containment and integrity constraints for xpath fragments. In *KRDB*, 2001.

[14] W. Fan, G. M. Kuper, and J. Siméon. A unified constraint model for xml. In *WWW '01*, pages 179–190, New York, NY, USA, 2001. ACM.

[15] P. Genevès and N. Layaïda. A system for the static analysis of xpath. *ACM Trans. Inf. Syst.*, 24(4):475–502, 2006.

[16] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of xml paths and types. *SIGPLAN Not.*, 42(6):342–351, 2007.

[17] P. Genevès, N. Layaïda, and A. Schmitt. Xpath typing using a modal logic with converse for finite trees. In *PLAN-X* [1], pages 61–72.

[18] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[19] T. X. Initiative. XUpdate - XML update language. `http://xmldb-org.sourceforge.net/xupdate/`.

[20] S. S. Ishtiaq and P. W. O'Hearn. Bi as an assertion language for mutable data structures. In *POPL '01*, pages 14–26, New York, NY, USA, 2001. ACM.

[21] C. Koch. Xpath leashed. In *PLAN-X* [1], pages 0–1.

[22] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL '01: Proceedings of the 15th International Workshop on Computer Science Logic*, pages 1–19, London, UK, 2001. Springer-Verlag.

[23] W3C. XML schema part 1: Structures second edition, Oct. 2004. `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/`.

[24] W3C. XML schema part 2: Datatypes second edition, Oct. 2004. `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/`.